Madeleine: An Efficient and Portable Communication Interface for RPC-based Multithreaded Environments

Luc Bougé, Jean-François Méhaut, Raymond Namyst

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Madeleine: An Efficient and Portable Communication Interface for RPC-based Multithreaded Environments

Luc Bougé — Jean-François Méhaut — Raymond Namyst

N° 3845

December 1999

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--3864--FR

RHÔNE-ALPES

Madeleine: An Efficient and Portable Communication Interface for RPC-based Multithreaded Environments

Luc Bougé*, Jean-François Méhaut*, Raymond Namyst*

Thème 1 — Réseaux et systèmes
Projet ReMaP

Rapport de recherche n° 3845 — December 1999 — 24 pages

Abstract: We introduce MADELEINE, a communication interface specifically designed to support distributed, multithreaded applications in both a portable and efficient way. Thanks to its new API, MADELEINE can implement RPC operations without any extra copy with respect to the underlying protocol. MADELEINE can thus achieve very good performance on a variety of protocols (MPI, TCP, BIP, VIA, etc.) and networks (Fast-Ethernet, Myrinet, SCI, etc.). MADELEINE serves as a building block for the multithread programming environment PM^2. We provide detailed experimental results for each representative implementation and discuss a number of possible improvements.

Key-words: Calcul haute performance, grappes, multithreading, interface de communication, protocole zéro-copie, RPC, accès mémoire distant.

This work has been partially supported by the French CNRS ARP Program on Architecture, Networks, Systems and Parallelism, and by the INRIA Cooperative Research Action ResCapA.

* Project ReMaP, LIP, ENS Lyon, 46, Allée d'Italie, 69364 Lyon Cédex 07, France. Corresponding author: Luc Bougé. Email: Luc.Bouge@ens-lyon.fr, Raymond.Namyst@ens-lyon.fr, Jean-Francois.Mehaut@ens-lyon.fr.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 —Télécopie +33 4 76 61 52 52

Madeleine : une interface de communication portable et efficace pour les environnements multithreads à base de RPC

Résumé : Nous présentons Madeleine, une interface de communication spécifiquement conçue pour supporter les applications multithreads distribuées de manière à la fois portable et efficace. Grâce à sa nouvelle interface de programmation (API), Madeleine peut implémenter efficacement les appels de procédures à distrances (RPC) sans copie supplémentaire par rapport au protocole sous-jacent. Madeleine peut ainsi réaliser des performances excellentes sur un large spectre de protocoles (MPI, TCP, BIP, VIA, etc.) et de réseaux (Fast-Ethernet, Myrinet, SCI, etc.). Madeleine est l'un des blocs de base de l'environnement de programmation multithread PM2. Nous présentons des résulats expérimentaux détaillés pour chaque implémentation représentative et nous discutons des améliorations possibles.

Mots-clés : High-performance computing, Cluster computing, Multithreading, Communication interface, Zero-copy protocol, Remote procedure call, Remote DMA.

Contents

1 Communication Interfaces for RPC-Based Multithreaded Environments **4**
- 1.1 Avoiding Extra Data Copies during Transmissions . . . 4
- 1.2 Designing a Portable Interface . . . 5
- 1.3 Coupling Thread Scheduling and Communication Progress . . . 5
- 1.4 What Do Existing Communication Interfaces Provide? . . . 5

2 The MADELEINE Communication Interface **6**
- 2.1 The Upper Programming Interface . . . 7
  - 2.1.1 Sending Messages . . . 7
  - 2.1.2 Receiving Messages . . . 8
  - 2.1.3 Basic Examples . . . 9
  - 2.1.4 Implementing a RPC operation . . . 10
- 2.2 The Lower Portability Interface . . . 11

3 Implementation and Performance **13**
- 3.1 MADELEINE over a Buffering Protocol: TCP . . . 13
- 3.2 MADELEINE over a Zero-Copy Protocol: BIP . . . 15
- 3.3 MADELEINE over the VI Architecture . . . 17
- 3.4 MADELEINE over the SCI Remote Memory Access Network . . . 19
- 3.5 MADELEINE as the communication interface of PM^2 . . . 20

4 Conclusion **21**

Introduction

Distributed multithreading is currently a popular technique to support the execution of parallel applications on MIMD machines. This is mainly due to the recent advent of distributed multithreaded environments [11, 21] and to their availability on a wide range of architectures. The distributed multithreaded approach has already been successfully applied to several application fields, such as combinatorial optimization, large-scale simulations [7], and molecular dynamics.

A large subset of these environments are based on *Remote Procedure Call* (RPC) communication API because their functionalities implicitly or explicitly rely on a mechanism able to invoke remote services (*e.g.*, RSR in Nexus [11], Light-weight RPC in PM^2 [21]). In contrast, distributed multithreaded environments are often implemented on top of portable *Send/Receive* communication interfaces such as PVM or MPI (*e.g.*, Athapascan [6]). Yet, it is clear that implementing a specialized, RPC-oriented communication API using a standard, handshake-oriented communication interface may not be the best way to go!

This amazing situation has often been considered as an unavoidable tradeoff between portability and efficiency. We claim that this is wrong by introducing a new communication interface called MADELEINE. The MADELEINE communication layer is designed to be portable across a large range of network architectures and protocols. The MADELEINE communication layer provides RPC-based multithreaded environments with *both* transparent and highly efficient communications. Specializing the communication interface is the key to escape from the tradeoff.

MADELEINE is organized in two layers. The low-level layer (Portability Interface) isolates the code that needs to be adapted for each targeted network protocol. The high-level layer (Programming Interface) provides communication functionalities including buffer management facilities. Thanks to the small number of architecture-dependent primitives, MADELEINE can be implemented on top of many network protocols, from standard, high-level ones (*e.g.*, TCP) down to specialized, low-level ones, such as BIP [23] for Myrinet, SBP [25] for Fast-Ethernet, SISCI for Dolphin SCI, etc. In this respect, our approach shares similarities with the Illinois Fast Message communication software.

This paper is organized as follows. We first revisit the communication needs of RPC-based multithreaded environments. The key objective is to avoid any additional copy in transmitting the arguments of the

RPC though their sizes are unknown to the receiver. Then, we introduce the MADELEINE communication interface: the high-level Programming Interface and the low-level Portability Interface. Then, we describe the implementation of MADELEINE on top of a number of representative protocols: standard TCP, the Gigabit BIP protocol for Myrinet, the Virtual Interface Architecture and Dolphin's SCI.

1 Communication Interfaces for RPC-Based Multithreaded Environments

A multithreaded environment is called "RPC-based" if most communications take place by means of remote invocations of services (also called Remote Procedure Calls). In this scheme of remote interaction, a *client* (usually a thread) sends a parameterized *request* (a function and a list of argument data) to a *server* (usually a process) which executes the specified *service* (calling the function with the arguments) to serve the request. According to the type of service executed, a response may be sent back to the client upon completion of the function. Many kinds of remote operations actually conform to this communication scheme. For instance, remote-read and remote-write functionalities which are provided by some distributed environments [10] are special cases of RPC operations. Thread migration [21, 29, 9] is another example: the parameter of the service is essentially the thread's stack and no response is expected on the calling side.

Most existing distributed multithreaded environments provide RPC-based mechanisms [11, 6]. The three following features determine their efficiency: zero-copy data transmissions, control over communication scheduling and portability. While the two last ones are not specific to RPC-based environments, the first requirement has a dramatic influence on the efficiency of RPC operations.

1.1 Avoiding Extra Data Copies during Transmissions

On a high speed network providing very low transmission latency (*e.g.*, Myrinet [2]), any extra message copy has a tremendous impact on performance [18]. Therefore, it is crucial that the underlying communication layer be able to ensure the delivery of messages without any extra copy of data. Many existing communication libraries actually provide such a functionality: those providing a very low level of network abstraction [27, 22, 23, 25], but also those – such as MPI – providing a rich set of high-level communication functionalities. However, implementing an RPC operation without any extra copy of data is more complex than merely implementing a zero-copy message transmission. Actually, when a RPC request is issued to a server, the server does not know the number and sizes of the arguments. This information has to be extracted from the request message itself. Only after this information is known can the server decide where these data should be stored. More specifically, the triggering of a zero-copy RPC operation involves three steps on the server side:

1. The request type is identified (*e.g.*, "Is this a migration or a remote put?") and some information is extracted from the network.
2. Then, actions are taken (for instance, dynamic memory allocations) to prepare for the receipt of the data.
3. Finally, the data are extracted from the network and directly stored at the right location in memory.

Efficiently implementing this scheme while avoiding extra copies of messages requires some knowledge of the underlying communication layer.

- Working on top of a MPI-like message-passing interface requires the client to send two messages instead of one. The first one carries the *header* of the request: its type, the number and sizes of the arguments, etc. The second one carries the *body* of the request, that is the regular data.
- In contrast, such a scheme can be efficiently implemented with a *single* message on top of a TCP-like communication layer, which buffers messages on the receiving side. The message carries the whole request (*header* + *body*), and its parts are successively extracted.

Observe that using a MPI implementation on top of TCP such as LAM-MPI prevents the user from gaining advantage of the buffering capability of TCP! The MPI interface is not flexible enough to allow efficient support of RPC operations. Hence, the programmer could be temptated to explicitly reflect in his high-level program some features of the low-level networking protocol. This is definitely *not* acceptable in terms of portability.

1.2 Designing a Portable Interface

Roughly speaking, the implementation of a distributed multithreaded environment essentially realizes the complex integration of a thread package and communication software. To be portable on a wide range of architectures, these two components must themselves be portable. As far as threads are concerned, the problem is not so critical since most available thread packages approximately provide the same functionalities. This makes it easy to build a common interface, actually similar to the POSIX-thread interface [15], which can be adapted to the targeted thread package in a straightforward manner.

In contrast, the situation is much more complex with respect to communication. Due to the variety of today's networking technologies (Ethernet, ATM, Myrinet, SCI, etc.), efficient communication libraries often provide low-level interfaces focusing on a particular network technology. As a result, their implementations are highly optimized for the underlying network hardware (BIP, SBP, SHMEM, etc.). Building complex applications such as multithreaded environments, directly on top of such interfaces leads to loosing portability. Large parts of the applications have to be specifically redesigned for each network hardware. This is the reason why so many people got involved in the design of portable communication interfaces (PVM [12], MPI [20]) which hide the network dependent features (reliability, flow control, message fragmentation, etc.) by providing a high level of abstraction.

1.3 Coupling Thread Scheduling and Communication Progress

Multithreaded applications are usually utmost sensitive to the policy used to schedule communication operations. This is especially true if the network is accessed in user-level space, as with optimized interfaces to high-speed networks such as BIP [23], LFC [1], SBP [25] or U-Net [26]. In this case, the application is neither signaled by the system nor by the hardware when a network event (such as the receipt of a packet) occurs. Instead, the application has to explicitly poll the network for incoming events.

Consider a thread waiting for the completion of a network operation. It repeatedly polls for the appropriate event. At each unsuccessful polling cycle, the thread should block and immediately yield the processor to another thread to overlap communication with computation. There is obviously a tradeoff between the application responsiveness and the overhead of useless polling actions [11, 17]. Hence, an efficient multithreaded environment should be able to monitor and tune its polling frequency. This requires the communication layer to be *thread aware* and to provide *explicit* polling operations.

1.4 What Do Existing Communication Interfaces Provide?

Many communication libraries have recently been designed to provide portable interfaces and/or efficient implementations to build distributed applications. They essentially fall into two classes. *High-level* communication libraries hide most underlying network features and provide advanced facilities to exchange data between processes (message passing [20]). *Low-level* communication libraries aim at getting the maximum performance out of the underlying hardware.

High-Level Interfaces PVM (Parallel Virtual Machine) is certainly one of the most popular "de facto standard" high-level communication libraries of the last decade. This is definitely due to its portability and ease of use, rather than to its performance! On this point indeed, PVM is overtaken by communication interfaces able to avoid buffering of messages, such as MPI. Some multithreaded environments have been implemented on top of PVM (the original release of PM^2 [21] and TPVM [10]), but their performance on

high-speed networks are poor, except for vendor specific versions such as PVMe on the Cray T3D. This is due to the (many) extra copies of messages made by the library.

MPI (Message Passing Interface), the standard communication interface proposed by the MPI Forum, is not subject to several of the PVM limitations. Its implementations usually transmit messages without extra copies. However, this implies that the receiver knows *in advance* the type and size of the expected data. This is clearly not the case in RPC-based environments, as discussed in Section 1.1. Athapascan [6], a multithreaded environments built on top of MPI, suffers from this drawback.

Low-Level Interfaces As high-level interfaces cannot meet the needs of time-critical applications, several research teams have designed low-level communication interfaces that can deliver much of the underlying hardware's performance. Such interfaces are then used as a basis to build high-level communication libraries (MPI-FM [18]). Although most of these libraries are message-passing oriented, they provide different interfaces and various levels of quality of service (reliability, flow-control, etc.).

BIP (Basic Interface for Parallelism [23]), probably today's most efficient way to transmit data over a Myrinet network, allows arbitrarily long messages to be sent but does not provide reliability nor flow-control. SBP (Streamlined Buffer Protocol [25]) and U-Net [26], which are highly optimized communication layers for Ethernet and Fast-Ethernet networks, provide some form of reliability and flow-control but force the upper layers to deal with preallocated fixed-sized buffers for message manipulation.

The Virtual Interface Architecture (VIA [8]) provides an abstract specification of the interactions between the operating system and the user-level communication level. A Virtual Interface (VI) consists of a pair of message queues: a send queue and a receive queue. The user program posts emission or reception requests on the message queues to send or receive data. A request consists of a descriptor describing the work to be done by the network interface controller. It includes a control segment and variable number of data segments (scatter-gather descriptors). Each queue manages a separate linked-list of descriptors. VIA also provides a completion queue construct used to link completion notifications from multiple work queue to a single queue.

An implementation of a multithreaded environment directly built on top of one of these communication interfaces could hardly be portable to another. The implementation of the Nexus multithreaded environment (Argonne, USA [11]) is based, as far as communications are concerned, on a generic interface that could adapt to all of these low-level libraries. However, this interface does not provide any sufficiently fine control over the underlying network protocols, so that the messages have to be buffered on the receiving nodes.

Medium-Level Interfaces The FM (Illinois Fast Messages) communication interface [22] is derived from the AM (Active Messages) communication software designed at Berkeley [27]. It could be considered as a *medium-level* communication layer as it can be ported on top of the previously mentioned low-level ones. Its interface provides a very simple mechanism to send data to a receiving node. This node is notified upon arrival by the activation of a handler. Release 2.0 of this interface provides some interesting gather/scatter features by introducing the concept of a "streaming message". This enables sending and receiving compound data made of several (not necessarily equal) pieces. This feature could allow an efficient implementation of zero-copy RPC operations as presented in Section 1.1. However, this streaming functionality is too general, and the efficiency would not be optimal on networks such as Myrinet: each segment would require a separate acknowledgment to enforce flow-control. The communication interface we propose in the next section does not have this drawback and provides a higher level of abstraction to the upper layers.

2 The MADELEINE Communication Interface

The MADELEINE communication interface [3] has been designed to meet the requirements of RPC-based multithreaded environments of *both* efficiency and portability. Unlike environments like MPI or PVM, MADELEINE is not designed to be directly used by regular user-level applications. This interface is especially targeted at RPC-based multithreaded environments such as PM^2 or Nexus. In this context, it is intended

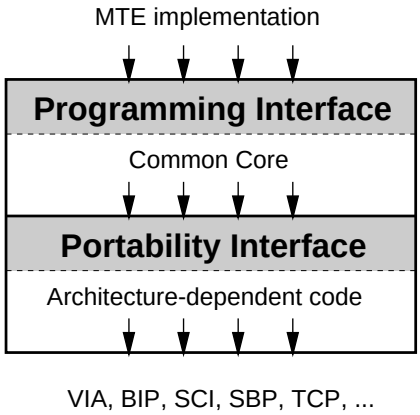

Figure 1: Structure of the MADELEINE communication interface.

to bridge the gap between the functionalities provided by low-level network protocols (such as BIP [23] or VIA [8]) and the requirements of high-level abstractions (such as the RPC mechanism).

MADELEINE has been designed to allow the upper software layers (*i.e.*, inside the multithreaded environment) to avoid extra copies of transmitted data. At the lowest level, this cooperation is realized using up-calls on the receiving side to allow upper layers to interactively participate in the reception and extraction of the data. At the user interface level, this cooperation is elegantly accomplished through the use of simple buffer management primitives.

MADELEINE is composed of two software layers: a generic user programming interface and a low-level portability interface whose implementation is network-dependent (Fig. 1).

2.1 The Upper Programming Interface

The MADELEINE Programming Interface provides a small set of primitives to build RPC-like communication schemes. Theses primitives look like classical message-passing oriented primitives, and they actually only differ in the way the reception of data is handled. Essentially, this interface provides primitives to send and receive *messages*, and several *packing* and *unpacking* primitives which allow the user to specify how data should be inserted into/extracted from messages.

A message may consist of several pieces of data, located anywhere in the user space. They are incrementally assembled (resp. disassembled) using the *packing* (resp. *unpacking*) primitives. This way, a message can be built (resp. examined) at multiple software levels, which allows for instance the use of piggy-backing techniques without losing efficiency. As a motivating example, consider a remote procedure call which takes an array of unpredictable size as a parameter. When the request reaches the destination node, the header is examined both by the multithreaded runtime (to allocate the appropriate thread stack and then to spawn the server thread) and by the user application (to allocate the memory where the array should be stored). Both software layers actually cooperate in the packing/unpacking of message data.

2.1.1 Sending Messages

Let us now describe how a user program prepares and sends a message to a remote node (Fig. 2). Such an operation starts by requesting MADELEINE to allocate a new *message descriptor* for a given destination. Then, a series of *packing* operations are performed to *register* the successive parts of the message. Eventually, the actual emission operation is requested.

```
sendbuf_init(dest_node);

   sendbuf_pack(...);
   sendbuf_pack(...);

sendbuf_send();
```

Figure 2: A typical send operation.

```
void handler()
{
  recvbuf_unpack(...);
  recvbuf_unpack(...);

  recvbuf_receive_data();
}

recvbuf_receive(handler);
```

Figure 3: A typical receive operation.

The critical point of a send operation is obviously the series of *packing* calls. Such packing operations simply *virtually* append some data to the message under construction. Thus, it may be the case that the data is not physically copied, but only registered as part of the message.

The prototypes of these functions are very similar to the buffer management primitives provided by the PVM interface (*e.g.*, `pvm_pkint`, `pvm_pkfloat`, etc.). For instance, the function used to append one or more integers to the current message has the following prototype:

```
void sendbuf_pack_int(int mode, int *elems, int nb);
```

Parameter `elems` is the address of an array of integers and `nb` stores the size of this array. The `mode` parameter plays an important role in the MADELEINE interface. It can be assigned different values. These modes define the actual semantics [4] of the `sendbuf_pack` operation. They are mutually exclusive:

`SND_SAFER` This mode indicates that MADELEINE should pack the data in a way that further modifications to the corresponding memory area should not impact on the message. In other words, an explicit *copy* is required. This is particularly mandatory if the data location is to be reused before the message is actually sent.

`SND_LATER` This mode indicates that MADELEINE should not consider accessing (*i.e.*, copying, or even sending) the value of the corresponding data until the `sendbuf_send` primitive is actually called. This means that any modification of the data between their packing and their sending shall actually update the message contents. This is typically useful when a message is to be sent to various destinations with only minor variations. In this case, the data can be packed only once and their values can be updated between the send operations.

`SND_CHEAPER` This is the default mode. It allow MADELEINE to do its best to handle the data as efficiently as possible. The counterpart is that no assumption should be made about the way and the point in time MADELEINE will actually access the data. Thus, the corresponding data should be left unchanged until the send operation has completed. We describe in Section 3 how this can be implemented according to the underlying network protocol. Note that most data transmissions involved in usual parallel applications accommodate the `SND_CHEAPER` semantics.

Of course, packing the same data multiple times with different semantics within the same message is forbidden. There is no other particular restriction on the order or on the number of packing operations with respect to their semantics. The only overall constraint imposed by MADELEINE is that the sequence of packing operations has to be exactly *mirrored* on the receiving side, as seen in the next section.

2.1.2 Receiving Messages

Receiving a message is always done in three steps (Fig. 3). First, the arrival of the message is detected (inside the `recvbuf_receive` function), which triggers the execution of a handler. Second, a number of unpacking operations (`recvbuf_unpack_*`) are performed within the handler to specify where data should be stored

```
char *data;
int i, size;

   sendbuf_init(dest_node);
   for(i=0; i<2; i++) {
      data = generate_str(); size = strlen(data)+1;
      sendbuf_pack_int(SND_SAFER | RECV_EXPRESS, &size, 1);
      sendbuf_pack_byte(SND_CHEAPER | RECV_CHEAPER, data, size);
   }
   sendbuf_send();
```

Figure 4: Sending a sample message with the MADELEINE Programming Interface

in memory. Third, a *commit* operation is executed (`recvbuf_receive_data`) to force the completion of the data transmission.

The last two steps need further explanations. As the *packing* operations, the *unpacking* ones also require a special parameter which specifies *when* the data should actually be received. There are two possible values for this parameter:

`RECV_EXPRESS` This mode forces MADELEINE to guarantee that the corresponding data is *immediately* available upon the completion of the *unpacking* operation. Typically, this mode is mandatory if the extracted data is to be used before the `recvbuf_receive_data` primitive is called. On some network protocols, this functionality may be available for free. On some others, it may induce a high penalty in latency and bandwidth. Data should be so extracted only when necessary. In the example described in Section 1.1, the message *header* would have to be unpacked this way.

`RECV_CHEAPER` This is the default mode. It allows MADELEINE to possibly defer the extraction of the corresponding data until the execution of `recvbuf_receive_data`. Thus, no assumption can be made about the exact moment at which the data will be extracted. Depending on the underlying network protocol, MADELEINE will do its best to minimize the overall message transmission time.

If combined with `SND_CHEAPER`, this mode guarantees that the corresponding data is transmitted as efficiently as possible. On high-performance network protocols such as VIA or BIP for instance, data will actually be transmitted without any extra copy.

For efficiency reasons, the way data is to be extracted from a message should be known by MADELEINE on the sending side. That is, on packing data a message, the programmer has to specify not only the sending mode, but also the receiving mode (for clarity, we omitted this detail in the previous section). Furthermore, the reverse is also true for the same reason: on unpacking data out from a buffer, the sending mode has to be specified in addition to the receiving mode (by logically *or*-combining the modes). This constraint could have been avoided at the price of a significant performance drop by letting the messages be self-describing, but we disregarded this solution. Because MADELEINE internal messages are not self-describing, *packing* and *unpacking* operations have to be done in the same order (matching pack and unpack operations must have the same rank within each series).

2.1.3 Basic Examples

The following example illustrates the power of the MADELEINE interface. Consider sending a message made of two strings, which are dynamically allocated by a function (`generate_str`) returning their address in memory. Their lengths are unpredictable. On the receiving side, one has first to extract the respective size of each string (an integer) before extracting the string themselves: the destination memory has to be dynamically allocated. The message shall thus contain four elements: two integers and two strings. The constraint is that each integer must be extracted `RECV_EXPRESS` *before* the corresponding string.

```
char *data[2];

void handler()
{
   int i, size;

   for(i=0; i<2; i++) {
      recvbuf_unpack_int(RECV_EXPRESS | SND_SAFER, &size, 1);
                         /* 'size' is available immediately */
      data[i] = malloc(size);
      recvbuf_unpack_byte(RECV_CHEAPER | SND_CHEAPER, data[i], size);
                  /* Caution: data is *not* yet available here. */
   }
   recvbuf_receive_data();
}

int main_function()
{
   recvbuf_receive(handler); /* The current thread will block and  */
                             /* wait for the arrival of a message. */
   printf("I received the strings: %s and %s\n", data[0], data[1]);
}
```

Figure 5: Receiving and inspecting a message with the MADELEINE Programming Interface

On the sending side, the code may look like the one listed on Figure 4. Note that the integers are packed using the `SND_SAFER` mode because the `size` variable is reused inside the loop.

Figure 5 displays the corresponding reception code. Specifying the `RECV_CHEAPER` mode for the transmission of strings may allow MADELEINE to send both objects without extra copies within a single network transaction, for instance with a gather/scatter mechanism. This is a major advantage over communication interfaces such as FM [22], which do not allow the user to express such a semantics: either a extra copy or an extra network packet would have to be sent in such a case.

2.1.4 Implementing a RPC operation

We now compare the implementation of a typical remote procedure call operation on top of MADELEINE and MPI. Actually, the RPC operation we describe below is the basis of the *ping-pong* test program used to estimate network performance of MADELEINE in Section 3. In this program, one-sided RPC operations are alternately triggered by two nodes to perform ping-pong communications. The unique argument of the RPC operation is a parameterized array of bytes. Handling a RPC consists in "receiving" the array and sending it back immediately to the sender. Figure 6 shows the skeleton of the program for reception. The emission code is straightforward.

The top of the figure lists the code of the MPI version of the ping-pong program. First, a receive operation is posted, waiting for the *header* describing the RPC itself (type, sender, etc.). Once this header has been received, the request is identified and the corresponding handler is executed (`handle_pingpong_rpc`). Within the handler, the argument of the RPC, that is, the *body* of the message, is retrieved through a second call to the `MPI_recv` primitive. As the ping-pong program was only designed to benchmark network performance the array is stored in a static global variable; in a real program, it would probably be dynamically allocated through a call to the `malloc` function. Note that the two `MPI_recv` operations cannot be merged in a single one because the size of the reception buffer for the second message is only known *after* the reception of the first message. Using a single MPI message only would not allow to directly receive the body at the right place.

The bottom of the figure lists the MADELEINE version of the corresponding code. The major difference is that there is conceptually only one receive operation. The header of the request is extracted with an immediate *unpack* operation (`RECV_EXPRESS`) and the argument is extracted with second, potentially deferred

Function	Operation
`init(configSize, taskIDs)`	Connections setup
`exit()`	Connections shutdown
`send_post(dest, iovec, count)`	Post the sending of msg to node 'dest'
`send_poll(dest)`	Check if msg sending is completed
`recv_header_post(func)`	Ready to receive a header
`recv_header_poll()`	Check if header received
`recv_body_post(exped, iovec, count)`	Ready to receive data
`recv_body_poll(exped)`	Check if data received

Table 1: The MADELEINE Portability Interface.

unpack (`RECV_CHEAPER`). Thanks to this scheme, MADELEINE is able to optimize the transfer of the array of bytes according to the underlying network. For instance, a zero-copy transfer is possible if the underlying network protocol provides the appropriate feature.

2.2 The Lower Portability Interface

The *Portability Interface* of MADELEINE is intended to provide a common interface to the various communication subsystems on which it may be ported. The design of this interface is critical because it has to be independent of any particular protocol, while achieving efficient performance on top of any of them.

The execution model of MADELEINE is based on the Active Messages (AM) model [27]. It is essentially a message-passing protocol, where exchanges are done between traditional processes. This means that although its implementation has to be thread-safe, the protocol does not need to be thread-aware. As a consequence, the upper layers have to ensure that only one thread is processing a receive operation at a time.

At this low level, a message consists of a set of one or more vectors. Each vector is a contiguous area of memory defined by a pair (*address*, *size*), where the size is expressed in bytes. The first vector of a message has a particular semantics with respect to the receive operation: this vector contains the data that must be extracted prior to the rest of the message (*i.e.*, the *header*). Once extracted, these data are immediately made available to the upper layers so that some application-level actions may be executed before the rest of the message is extracted.

Figure 7 sketches a situation where Process A is sending a message to Process B. First, the message *header* (*i.e.*, the first vector) is sent to Process B (Step 1). On its receipt, a *handler* is executed with the header as a parameter (Step 2). This handler, which is defined by the upper layer, can inspect the header and take appropriate actions to prepare for full message extraction. Typically, the handler may compute the locations where the data should be placed (Step 3). Once the handler has completed, an acknowledgment is sent back (Step 4) and Process A is allowed to send the vectors of remaining data (*i.e.*, the message *body*, Step 5) . These data are directly stored at the right memory locations on their receipt (Step 6).

We emphasize that this scenario is a conceptual description of the protocol. In fact, specific implementations may well diverge from this scheme to get the best performance out of the underlying layer. We discuss more precisely such implementation issues in Section 3.

The Portability Interface we propose is composed of only 8 function prototypes (Table 1) which represent the architecture-dependent primitives of MADELEINE. Two of them (`init` and `exit`) deal with the management of connections respectively at the beginning and at the end of the execution of an application. The remaining 6 primitives deal with communication.

Sending Messages As MADELEINE is intended to be used in a multithreaded context, operations that would ordinarily block the calling process (*e.g.*, the sending of a message) should only block the calling thread. However, the portability interface has no knowledge about the thread package which is actually used in the upper layers.

```
char body[MAX_SIZE];

void handle_pingpong_rpc(header_t *header)
{
   MPI_recv(body, header->size, MPI_BYTE, header->sender, BODY_TAG,
            MPI_COMM_WORLD);
   ... /* execute the same RPC back on the sender (it's a ping pong) */
}

int main()
{
   header_t header;

   MPI_recv(&header, sizeof(header), MPI_BYTE, MPI_ANY_SOURCE, HEADER_TAG,
            MPI_COMM_WORLD);
   switch(header.request_type) {
      case PING_PONG : handle_pingpong_rpc(&header);
      ...
   }
}
```

```
char body[MAX_SIZE];

void handle_pingpong_rpc(header_t *header)
{
   recvbuf_unpack_byte(RECV_CHEAPER | SND_CHEAPER, body, header->size);
   recvbuf_receive_data();
   ... /* execute the same RPC back on the sender (it's a ping pong) */
}

void generic_handler()
{
   header_t header;

   recvbuf_unpack_byte(RECV_EXPRESS | SND_CHEAPER, &header, sizeof(header));
   switch(header.request_type) {
      case PING_PONG : handle_pingpong_rpc(&header);
      ...
   }
}

int main()
{
   recvbuf_receive(generic_handler);
}
```

Figure 6: Implementing a RPC operation on top of MADELEINE (top) and on top of MPI (bottom).

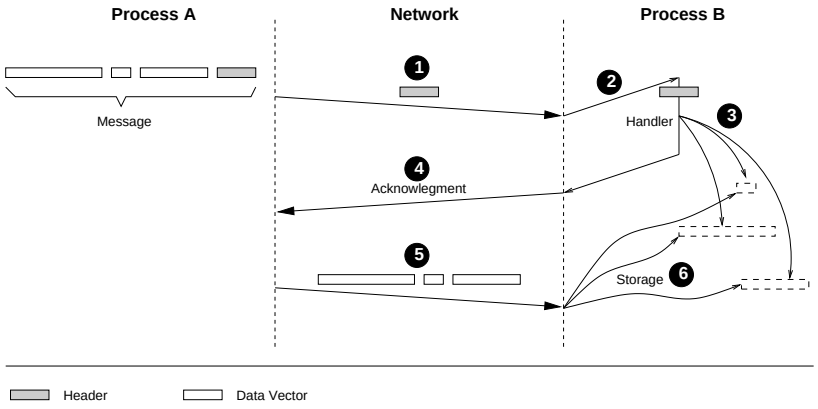

Figure 7: Conceptual view of the data-exchange protocol in the Portability Interface.

Therefore, a sending operation is done in two steps. A call to the `send_post` primitive initiates the sending of a message and returns without blocking. The destination process and the message (an array of Unix standard `iovec` structures) are given as parameters. Then, further calls to `send_poll` have to be made until the primitive returns `TRUE`, which means that the message transmission has completed on the sending side.

Receiving Messages To receive a message, the first step is to receive the message *header*, which is done by a call to `recv_header_post(handler)` followed by one or more calls to `recv_header_poll()`. As soon as the header is completely received, the call-back function *handler* gets executed: the control is temporarily yielded to the upper layers. The idea is that the handler has to inspect the header, to prepare for the receipt of the message *body* and to actually extract the data from the network. The second step typically involves building an appropriate array of vectors. The last step is realized by calling `recv_body_post` with the array as a parameter, followed by subsequent calls to `recv_body_poll` until completion.

3 Implementation and Performance

The MADELEINE interface is designed to be portable, to provide efficient RPC-like communications and to minimize the overhead introduced over the underlying network protocol. We illustrate these properties by describing the implementation and performance of MADELEINE over some representative protocols. In spite of the variety of their features, the overhead added by MADELEINE is small and constant in all cases.

All the experiments reported in this paper have been carried out on our local cluster of PC: a number of Intel PentiumPro 200 MHz nodes, with 64 MB of memory, run under Linux, interconnected by a variety of networks and protocols.

3.1 MADELEINE over a Buffering Protocol: TCP

On transmitting a message, the *buffering* protocols store data in some temporary place on the destination process. Thus, the effective receipt of these data can be deferred, and the message can be read in several steps. Of course, such protocols are *not* zero-copy: the best MADELEINE can do is not to add any extra copy to those made by the underlying layer. The TCP/IP transport protocol belongs to this category, as well as protocols such as SBP/Fast-Ethernet.

The implementation of MADELEINE over TCP is straightforward. The full message (*header* + *body*) is simply sent as a stream of bytes which is regulated by the internal TCP flow control algorithm. The sending

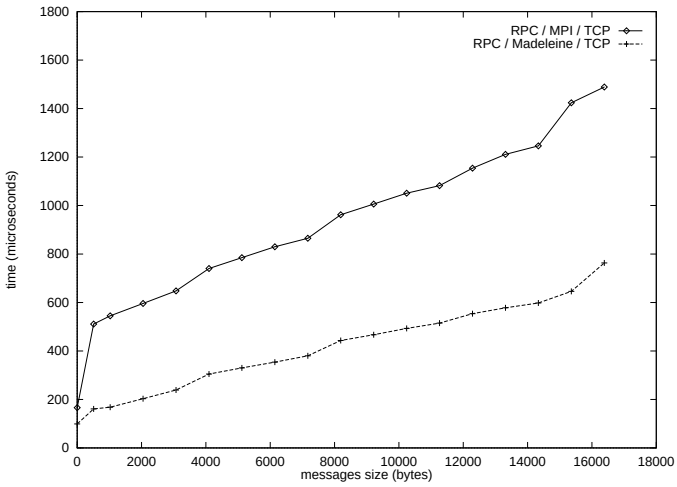

Figure 8: Performance of Remote Procedure Calls over MADELEINE and over MPI. The underlying communication protocol is TCP/IP on top of a 100 Mb/s Fast-Ethernet network.

primitives of the Portability Interface (see Table 1) are just implemented by calling the non-blocking `writev` Unix primitive.

On the receiving side, the detection of the message availability (`recv_header_poll`) is realized through a call to the Unix `select` primitive. Then, the message *header* is extracted by means of a `read` operation. Finally, the message *body* can be extracted with a `readv` operation. As a result, no extra message nor extra copy is introduced by MADELEINE on top of TCP. Observe that this scheme does obey the semantics of Figure 7, while its implementation is much simpler.

Performance of a complete RPC operation To evaluate the performance of our TCP implementation, we consider processing an RPC operation respectively over MADELEINE/TCP and over MPI/TCP, as described in Section 2.1.4. The parameter of the service function is is an array of bytes, whose length vary. Figure 8 reports the observed latency for various sizes of the array. The time is measured through a ping-pong procedure: it spans from the initialization of the communication on the source machine up to the *effective* execution of the function on the destination machine.

The gap in performance between MADELEINE and MPI is huge and proportional to message size: the latencies obtained with MPI are approximately twice as large as those obtained with MADELEINE. This demonstrates the better adequacy of the MADELEINE interface to support RPC operations on buffering protocols. This performance gap is due to the number of TCP messages exchanged within the MPI version. In fact, MADELEINE needs only one TCP message to process the RPC, whereas the MPI interface requires the application to send two *user* messages. In addition, the internal flow-control algorithm of the MPI implementation may even trigger the sending of a third message from the receiver back to the sender. It is important to note that this gap is in no way due to the *quality* of the MPI implementation. It simply reveals that the MPI interface lacks functionalities as far as RPC-like operations are concerned. Experiments featuring other buffering protocols (such as SBP [25]) lead to similar results.

Performance of a single message communication Although the advantage of using MADELEINE to build RPC-based multithreaded environment is obvious, it is interesting to evaluate the overhead introduced by MADELEINE over the underlying TCP protocol with respect to the communication of a single message.

Figure 9 displays the performance of raw message passing over TCP compared with the performance of RPC on top of MADELEINE. Although the comparison between remote procedure calls and raw message transfer is not really fair, it is clear that the two curves show that the overhead is actually constant and

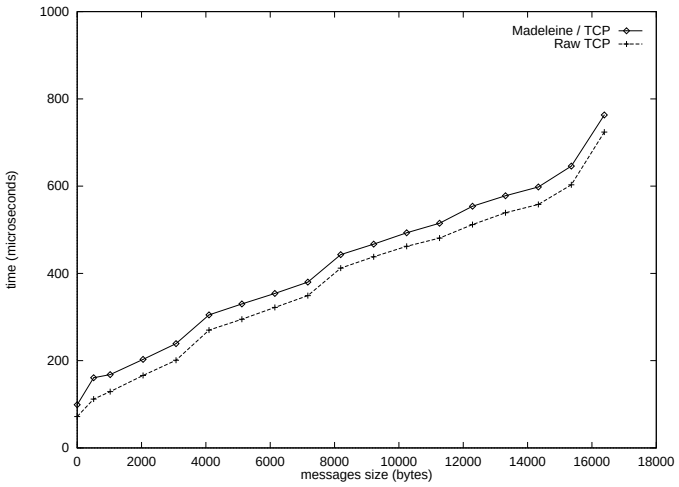

Figure 9: Overhead of MADELEINE on TCP/IP. The underlying network is 100 Mb/s Fast-Ethernet.

small (30 μs, to be compared with the 120 μs of latency for an empty message over TCP). Since only one TCP message is involved during a MADELEINE remote procedure call, this overhead is to the few additional data added into the *header* and to the software overhead of MADELEINE. As the overhead is small and independent of the message length, the maximum communication bandwidth reached by MADELEINE is close to that of the underlying protocol.

3.2 MADELEINE over a Zero-Copy Protocol: BIP

A communication protocol which can realize zero-copy transmissions obviously cannot provide any internal buffering of data. A message should actually be emitted *only* when the destination node is ready to handle it. Thus, a flow-control mechanism has to be set up between the sender and the receiver. Some communication protocols integrate such a mechanism internally, but some others leave this responsibility to upper layers.

BIP (Basic Interface for Parallelism [23]) is a very low-level communication interface dedicated to the Myrinet network. The major feature of BIP is to allow zero-copy message transmission between end-point processes. This is realized by providing direct access to the Network Interface Card (NIC) in user space. The operating system is not involved in communications and the performance of this communication layer is excellent: 5.5 μs of latency for 4-byte messages, with a maximum bandwidth of 126 MB/s. BIP provides no flow-control for messages of arbitrary length. Small messages may however be sent *before* the corresponding receive operation is posted. They are temporarily stored in a fixed-size buffer, but no flow-control is done to avoid overflowing this buffer.

The implementation of MADELEINE over BIP is more complex than over TCP, because of the lack of flow-control. Transmitting a MADELEINE message over BIP closely follows the scheme of Figure 7. The first BIP message carries the *header*; the second one informs the sender that the receiver is ready; the remaining ones carry the *body*. Since the the current version of the BIP interface does not allow the sending of non-contiguous data in a single message, more than three messages will be sent if the MADELEINE message is composed of several data vectors. To avoid buffer overflow on sending a succession of (small) messages carrying the MADELEINE *header*, a credit-based flow-control algorithm is added on top of the scheme.

All the primitives of the Portability Interface are directly implemented on top of BIP asynchronous operations, which fit exactly the semantics of the *post/poll* operations of MADELEINE. We also take advantage of the DMA engine of the Myrinet card. BIP intensively uses it to transfer data between the host's main memory and the NIC. The send and receive operations of MADELEINE can thus be highly overlapped with computation.

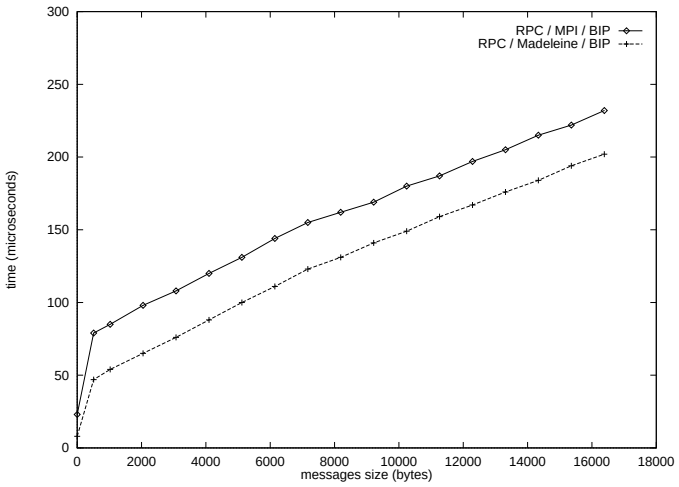

Figure 10: Performance of Remote Procedure Calls over MADELEINE and over MPI. The underlying communication protocol is BIP on top of a 1 GB/s Myrinet network.

Performance of a complete RPC operation As for TCP in Section 3.1, we consider the gain of using MADELEINE for the implementation of RPC operations instead of MPI. The MADELEINE and the MPI test programs remain unchanged, but their BIP implementations are used instead of TCP. The MPI implementation is derived from of MPICH [24], with additional optimization specific to BIP. The results are displayed on Figure 10.

The gap between MADELEINE and MPI is constant as expected. The MADELEINE implementation on top of BIP does approximately the same job as the implementation of MPICH: in both cases, three BIP messages are used to realize a single remote procedure call. The latency of null-RPC over MADELEINE and BIP is 8μs, to be compared with the 12μs of MPICH/BIP. It is due to the implementation of MPICH itself whose layered design introduces a considerable software overhead compared to MADELEINE.

Performance of a single message communication Unlike the TCP implementation, the BIP implementation of MADELEINE induces an overhead due to several factors:

1. Some additional data are transmitted, such as the tag of the request or some piggy-backed flow-control fields (four or five 32-bit words in practice).
2. Additional messages may be transmitted either to carry the previously mentioned data or to ensure flow control.
3. MADELEINE obviously introduces some software overhead to manage messages and handlers.

In fact, this overhead is independent of the size of the data. It is less than 10 μs on our platform.

Note however MADELEINE keeps much freedom to implement its communication scheme. This is used in the following amazing way. If the MADELEINE message to be sent is sufficiently small, then *both* the *header* and the *body* can be sent within a single BIP message. On the receiving side, this message is stored into a pre-allocated buffer. The *header* and the *body* are then selectively extracted by MADELEINE, very much as in the TCP implementation. Under these conditions, the overhead of MADELEINE becomes very low, and the latency measured for a null-RPC (*i.e.*, with no argument) using MADELEINE is only 8 μs on top of BIP. This is even better than sending a null message on top of MPICH/BIP!

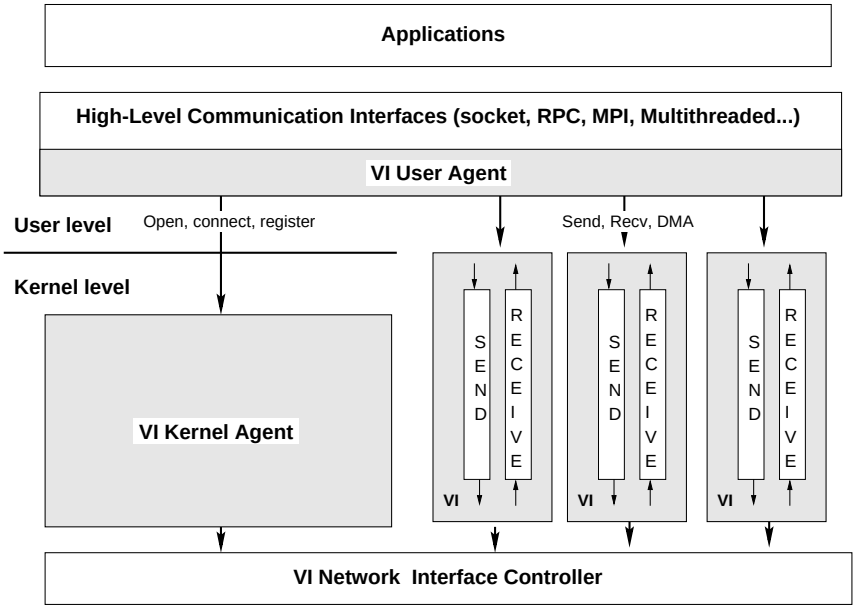

Figure 11: The Virtual Interface Architecture

3.3 MADELEINE over the VI Architecture

The research on communication interfaces has already demonstrated that a user-level approach [27, 22, 23] cannot be avoided to achieve high performance (minimal latency, maximal bandwidth). From this viewpoint, the model defined by VIA originates in some pioneers of user-level communication interfaces such as U-Net [26] or Active Messages [27]. From a technical point of view, one of the main contributions of VIA is to provide a clean specification of the interactions between the operating system and the user-level communication level. The VIA specification [8] is based on two main modules: 1) The *Kernel Module*, which initializes the communication environment and interacts with the memory management unit, 2) The *User Module*, which implements the user-level communication operations (send, receive). Note that the VIA specification is generic and independent of any operating systems. Actually, preliminary implementations are currently available on both Windows-NT and Unix (Linux) systems [28].

A *Virtual Interface* (VI) is the core concept of VIA (see Figure 11). It allows a user program to perform message passing operations. A VI consists of a pair of message queues: a *Send Queue* and a *Receive Queue*. The user program posts emission or reception requests on the message queues to send or receive data. A request consists of a descriptor describing work to be done by the network interface controller. It includes a control segment and variable number of data segments (scatter-gather descriptors). Each queue manages a separate linked-list of descriptors. VIA also provides a *Completion Queue* construct used to link completion notifications from multiple work queue to a single queue. A user process can own many VI, one for each point-to-point bidirectional connection. The kernel agent defines the mechanisms needed to establish the connections between nodes, in cooperation with the operating system.

Depending on the descriptor, data can be transmitted via regular message passing mechanisms or via Remote Direct Memory Access (RDMA) read/write operations. Before a descriptor is posted, all the data buffers it references should explicitly be *registered* in the VIA runtime.

MADELEINE is one of the first high-level communication software available on top of VIA. The implementation of the Portability Interface on top of this VI architecture is easy and efficient, because gather/scatter functionalities are provided with the descriptor management. Figure 12 displays the typical execution path during the sending of a message *body* formed out of three vectors. The MADELEINE generic message management code first builds an array describing the message. This array is passed to the Portability Interface

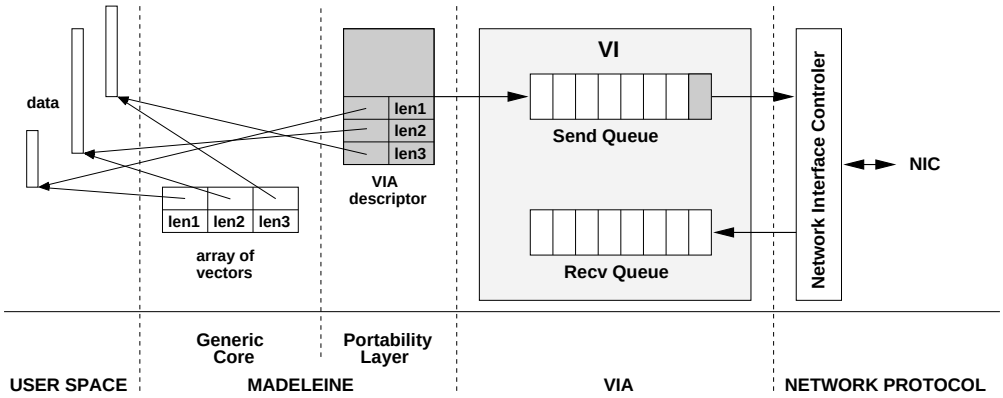

Figure 12: Sending data in the VIA implementation of MADELEINE.

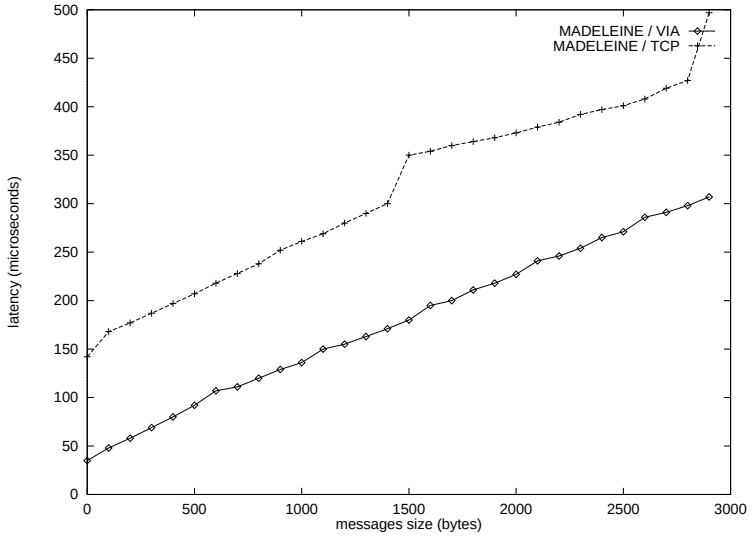

Figure 13: Performance of MADELEINE on top of M-VIA and TCP over a 100 Mb/s Ethernet network.

which is VIA-specific in this case. A single VIA descriptor is then built by merely copying the three entries of the array of vectors. At this point, the memory areas containing the user data are registered in VIA. Finally, the descriptor is posted into the Send Queue (which is an asynchronous operation) and the current thread can yield the processor until the operation is completed.

The implementation of a whole message sending is slightly more elaborate because VIA does not provide flow-control. Thus, before a descriptor one can be posted into a Send Queue, one has to make sure that a corresponding descriptor has already been posted in the destination Receive Queue. MADELEINE uses acknowledgment messages to control the posting of message bodies. As far as *headers* are concerned, a simple credit-based algorithm is used so that acknowledgment messages can be avoided in most cases.

We have used the M-VIA *software* implementation of VIA (developed at Berkeley Laboratories, `http://www.nersc.gov/research/FTG/via/`) on a Fast-Ethernet (FE) 100 Mb/s network between two Pentium 133 MHz nodes running Linux. Figure 13 shows the performance (latency) of raw message passing over MADELEINE using respectively M-VIA/FE and TCP/FE as underlying protocols. Measurements are obtained through a ping-pong test.

The gap between the two curves is significantly large, which confirms the interest of being portable on low-level protocols. We achieve a 37 μs latency with MADELEINE over VIA/FE with respect to 150 μs with MADELEINE over TCP/FE. The bandwidth achieved by MADELEINE over VIA/FE for 64 kB messages is over 11.5 MB/s, very close from the maximum 100 Mb/s figure.

3.4 MADELEINE over the SCI Remote Memory Access Network

The TCP and BIP protocols are strongly message-passing oriented. In contrast, the SCI (*Scalable Coherent Interface*) IEEE specification introduces a network technology providing remote addressing capabilities. When connected through a SCI-compliant network, a computer can allow part of its physical memory to be remotely accessed by processes running on other computers. When a "client" process maps such a remote memory in its virtual address space, this mapping actually redirects all memory accesses to the SCI network card through the system bus (*e.g.*, PCI for instance). Then, the network forwards the corresponding request to the target host's SCI card which will satisfy the requested operation (*e.g.*, `read`, `write`). Using such an SCI mapping, processes can transparently perform remote `read` or `write` operations by simply executing regular `load` and `store` processor instructions. In addition, SCI provides two explicit communication features. A Direct Memory Access (DMA) mechanism can be used to initiate asynchronous transfers to remote hosts. DMA operations do not involve the central processor during data transfers so that communication can be overlapped by computation. Also, the SCI protocol allows interrupts to be remotely triggered. This provides the basis to implement reactive communication layers such as Active Messages [27]. The implementation of MADELEINE on top of SCI uses Release 1.9.2 of SISCI driver interface developed by Dolphin ICS [13], which provides all the aforementioned functionalities. Currently, MADELEINE only uses remote write operations.

Implementing the MADELEINE portability layer on SCI basically means supporting message passing on SCI. To this end, several implementation alternatives might be considered [14]. However, the current Dolphin drivers introduce a major constraint that makes it generally impossible to avoid extra copies of transmitted data. The memory used as a target for remote read or write operations must be physically allocated within the driver by an explicit operation before being mapped in user space. This prevents regular memory areas such as global variables or data allocated in the heap, from being used for SCI remote operations. As a result, the user data have always to go through a "special" memory area on their way to the final destination.

Consequently, we have implemented a message passing mechanism based on remote message queues. To send a message (or a part of it), one places it into the message queue of the remote destination by mean of a remote write operation. It is then eventually extracted from this queue by a local read operation issued by the receiver. Each pair of processes is associated with two messages queues, one per process, in order to support bidirectional communications: $N \times (N-1)$ queues are used in a configuration of N processes.

Figure 14 shows more precisely how a message queue is managed. The main data structure is a circular FIFO buffer allocated on the receiver process and remotely mapped by the sender. The sender maintains a local *write pointer* which is remotely updated on the receiver side after a new message chunk has been sent. Symmetrically, the receiver maintains a local *read pointer* which is updated on the sender side as soon as a chunk has been extracted from the buffer. Because write ordering is not guaranteed on SCI, *memory barriers* have to be used to ensure that the various underlying caches and buffers (PCI write combining cache, SCI buffer) are flushed between writing a message chunk and updating the write pointer. Note that each process can thus safely avoid buffer overflows/underflows by simply comparing local values.

As expected, this implementation of message queues involves an extra copy of data on the receiving size. However, decomposing messages into *chunks* of adequate size generates a helpful pipeline [5].

Remote interrupts could have been used to signal the data transfer completions. We did not consider them because they introduce a huge overhead compared to the polling policy we use. However, in some situations, it can be shown that they perform better than polling [17, 19, 16]. We are currently investigating a mixed approach which would adaptively use interrupts on demand, when high reactivity is needed.

The same remark applies regarding DMA transfers. The overhead associated to starting the SCI DMA embedded engine is so high (typically, 200 μs) that it should only be used for large messages. We did not implement such a mechanism in the current version. However, we plan to integrate DMA facilities in the

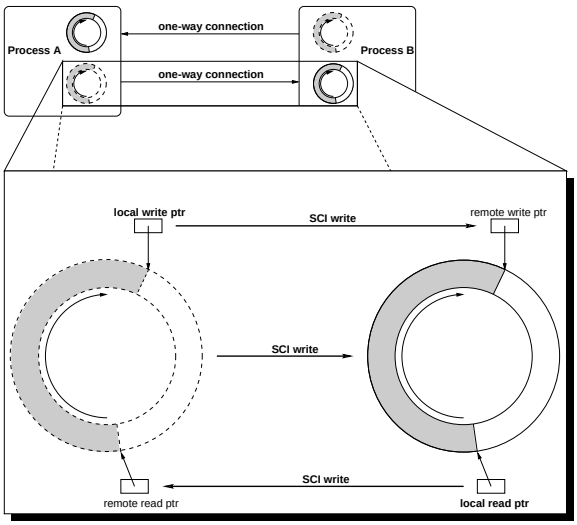

Figure 14: Two one-way SISCI remote mappings are established between each node. The solid buffer on the right is physically allocated on the receiving process. It is remotely mapped onto the emitter (dashed copy on the left).

next one. Not only it will increase the bandwidth for large messages, but it will also allow the overlapping of communication by computation when possible.

Performance of a single message communication Figures 15 and 16 display the performance achieved by MADELEINE on top of the SISCI low-level interface. The latency of MADELEINE for a null message is 5.4 μs, to be compared with the excellent 2.5 μs of SISCI. The asymptotic throughput of MADELEINE and SISCI are the same: 70 MB/s. It is interesting to observe that the throughput of SISCI is at its best for medium-size messages (74 MB/s for 100 kB–1 MB messages), and then gracefully decreases. This problem does not appear with MADELEINE, as MADELEINE messages are split into chunks of 8 kB or so.

One can observe that the overhead of MADELEINE over raw SISCI on Figure 15 is *not* constant. As mentioned above, MADELEINE cannot achieve a zero-copy data transfer on top of the current SISCI implementation, and an additional copy is needed to move data from the reception buffer to user space. The growing gap between the curves is mainly due to this copy. The additional overhead of MADELEINE apart from this copy is only 3 μs: it is constant, as expected.

3.5 MADELEINE as the communication interface of PM^2

MADELEINE was originally designed to serve as a building block for the distributed multithreaded environment PM^2 (Parallel Multi-threaded Machine) [21] on top of Linux. This environment manages lightweight threads in user space. A very large number of threads can be handled at the same time: the only limitation is the size of the available memory. The switching time between threads is less that 1 μs on our experimental platform.

In PM^2, a RPC consists of forking a remote thread to execute a specified service. A special case of this general mechanism is the *preemptive migration* of a thread between two (binary compatible) nodes. This is done in three steps. First, the thread is frozen by the scheduler, and both its descriptor and the useful part of its stack are packed in a buffer; then, the buffer is sent to the destination module; and last, the thread's descriptor and stack are unpacked on the destination module, the stack is relocated in a new address space, and the thread is unfrozen.

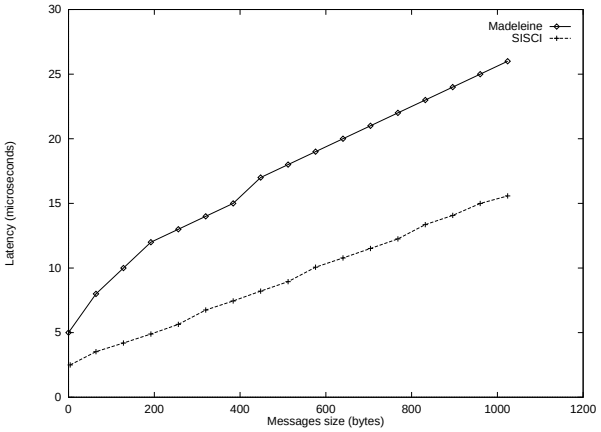

Figure 15: Latency of MADELEINE compared to the raw latency of SISCI.

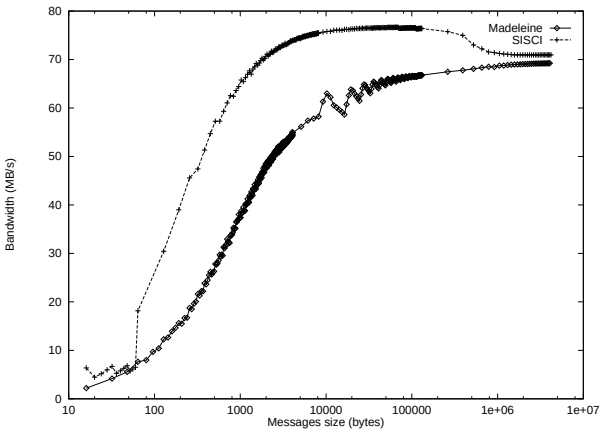

Figure 16: Throughput of MADELEINE compared to the raw throughput of SISCI using load/store operations.

The performance of thread migration is highly dependent on the performance of the underlying RPC. It can thus be seen as a significant benchmark for the communication interface. In this experiment, we consider a simple PM^2 program with only one thread with minimal stack size (650 bytes). This thread is created on the first node, then migrated onto the second node and migrated back onto the initial one. This forth-and-back sequence is executed 1000 times on our platform, and the average migration time is computed. The most significant figures reported can be listed as follows:

Protocol/Network	Migration time
SISCI/SCI	25 μs
BIP/Myrinet	57 μs
MPI/BIP/Myrinet	79 μs
TCP/Fast-Ethernet	245 us
MPI/TCP/Fast-Ethernet	330 μs

As expected, the best results are observed with high-speed networks such as SCI. Because the latency for short messages with SCI is very low, the migration of small threads on top of SCI and SISCI is as short as 25 μs. To the best of our knowledge, this sets a new world record for thread migration. The previous record was held by University of Berkeley and Active Thread [29] project with 150 μs for a thread migration on top of Myrinet and Active Messages [27]. It is difficult to compare the results with different networks but the figures are significantly improved with PM^2. On top of Myrinet, thread migration latency is 57 μs with the BIP communication protocol and 79 μs with an optimized implementation of MPI on top of BIP and Myrinet. The figures for Fast Ethernet are well behind (245 μs and 330 μs) because the communication protocol (TCP) is implemented at the operating-system level and the network technology provides less performance.

4 Conclusion

The goal of MADELEINE is to provide distributed, multithreaded applications with a *both* portable and efficient communication interface. Such applications crucially rely on *Remote Procedure Call* Procedure Call (RPC) communications, instead of simple message exchanges, which places a challenging constraint on the design of the interface. For instance, high-level interfaces such as MPI cannot achieve good performance for RPC operations: their design is not flexible enough. On the opposite, very good performance can be achieved by implementing RPC at the lowest level of the network interface. But such an implementation is

obviously highly dependent on the specific features of the interface, and no portability may be expected. We believe that MADELEINE provides a good trade-off between these extreme approaches.

The following table reports our best figures for sending a null message with MADELEINE in the `SND_CHEAPER` mode.

Protocol/Network	Latency	Bandwidth
SISCI/SCI	6 μs	70 MB/s
BIP/Myrinet	8 μs	125 MB/s

The overhead of MADELEINE is very low, of the same order of magnitude as the current network-specific low-level interfaces. As of today, MADELEINE is available on the following protocols:

- BIP (Myrinet),
- SISCI (Dolphin SCI),
- VIA on top of Fast-Ethernet (through the M-VIA Berkeley implementation),
- TCP,
- MPI and PVM.

The source code for all these implementations can be found at URL `http://www.ens-lyon.fr/~rnamyst/pm2.html`. Some other implementations under development can also be found at this address.

A major weakness of the design of MADELEINE as reported in this paper is that the interface is strongly oriented towards message passing on a homogeneous, fixed network of nodes. In fact, this bias is inherited from the design of PVM/MPI, from which MADELEINE evolved. It is obviously not well-suited to the recent network technologies such as SCI or VIA based on the notion of *remote DMA* operations. The design of the interface does not allow to take advantage of all the features of such networks: a RPC operation with a small body could probably be best handled by copying the service number and its arguments onto the destination host though a direct remote DMA operation, instead of packing them into a message. Also, this implementation of MADELEINE does not allow to modify the network configuration dynamically. It is not possible to add or delete nodes. Again, this is inherited from the original design of MPI.

With the advent of meta-computing, supporting heterogeneous, dynamically evolving networks of nodes becomes of primary importance. In the case of MADELEINE, it would be for instance interesting to provide a uniform communication interface for clusters of high-performance clusters: for instance, a Myrinet cluster of PC and a SCI one, interconnected through a TCP/Giga-Ethernet link. This would require MADELEINE to handle multiple network interface at the same time. Also, the ability to adapt to dynamically changing configuration would be useful. These features will eventually be included in the new version of the MADELEINE interface, called MADELEINE 2, currently under development.

References

[1] R. Bhoedjang, T. Ruhl, and H. Bal. Design issues for user-level network interface protocols on Myrinet. *IEEE Computer*, 31(11):53–60, November 1998.

[2] N. Boden, D. Cohen, R. Feldermann, A. Kulawik, C. Seitz, and W. Su. Myrinet: A Gigabit per second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[3] Luc Bougé, Jean-François Méhaut, and Raymond Namyst. Madeleine: an efficient and portable communication interface for multithreaded environments. In *Proc. 1998 Int. Conf. Parallel Architectures and Compilation Techniques (PACT '98)*, pages 240–247, ENST, Paris, France, October 1998. IFIP WG 10.3 and IEEE.

[4] Luc Bougé, Jean-François Méhaut, and Raymond Namyst. Efficient communications in multithreaded runtime systems. In *Parallel and Distributed Processing. Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, volume 1586 of *Lect. Notes Comp. Science*, pages 468–182, San Juan, Puerto Rico, April 1999. IPPS/SPDP 1999, Springer-Verlag.

[5] Luc Bougé, Jean-François Méhaut, Raymond Namyst, and Loïc Prylli. Using the VI Architecture to build distributed, multithreaded runtime systems: a case study. In *Proc. 2000 ACM Symposium on Applied Computing (SAC '2000)*, Como, Italy, March 2000. ACM SIGAPP, ACM. To appear.

[6] J. Briat, I. Ginzburg, M. Pasin, and B. Plateau. Athapascan runtime: Efficiency for irregular problems. In *Proceedings of the Euro-Par '97 Conference*, volume 1300 of *Lect. Notes Comp. Science*, pages 591–600, Passau, Germany, 1997. Springer-Verlag.

[7] S. Brunett and T. Gottschalk. A large-scale metacomputing framework for the ModSAF real-time simulation. *Journal of Parallel Computing*, 24(12):1873–1900, 1998.

[8] Compaq, Intel, and Microsoft Corporations. *Virtual Interface Architecture. Version 1.0*, December 1997. Available from `www.viarch.org`.

[9] B. Dimitrov and V. Rego. Arachne: a portable threads systems supporting migrant threads on heterogeneous network farms. *IEEE Transactions on Parallel and Distributed Systems*, 9(5):459–469, May 1998.

[10] A. Ferrari and V. Sunderam. TPVM: Distributed concurrent computing with lightweight processes. In *Proc. of IEEE Conf. High Performance Distributed Computing*, pages 211–218, Washington, DC, August 1995.

[11] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal on Parallel and Distributed Computing*, 37:70–82, 1996.

[12] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*, 1994.

[13] F. Giacomini, T. Amundsen, A. Bogaerts, R. Hauser, B. Johnsen, H. Kohmann, R. Nordstrom, and P. Werner. *Low-level SCI software functional specification*, May 1998. Esprit Project 23174: Software Infrastructure for SCI (SISCI).

[14] M. Ibel, K. Schauer, C. Scheiman, and M. Weis. High-performance cluster computing using Scalable Coherent Interface. In *Seventh International Workshop on SCI-based Low-cost/High-performance Computing (SCIzzL-7)*, Santa Clara, California, March 1997.

[15] IEEE Standards Department. *1003.4d8 POSIX System Application Program Interface: Threads Extensions [C language]*, 1994.

[16] K. Langendoen, R. Bhoedjang, and H. Bal. Models for asynchronous message handling. *IEEE Concurrency*, 5(2):28–38, April 1997.

[17] K. Langendoen, J. Romein, R. Bhoedjang, and H. Bal. Integrating polling, interrupts, and threads management. In *Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computations (Frontiers '96)*, pages 13–22, Annapolis, Maryland, October 1996. IEEE.

[18] M. Lauria and A. Chien. MPI-FM: High performance MPI on workstation clusters. *Journal on Parallel and Distributed Computing*, 1(40):4–18, January 1997.

[19] O. Maquelin, G. Gao, H. Hum, K. Theobald, and X. Tiam. Polling watchdog: Combining polling and interrupts for efficient message handling. In *Proc. of the 23rd International Symposium on Computer Architecture*, pages 179–190, New York, May 1996. ACM Press.

[20] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, March 1994. Available from `www.mpi-forum.org`.

[21] Raymond Namyst and Jean-François Méhaut. PM2: Parallel multithreaded machine. a computing environment for distributed architectures. In *Parallel Computing (ParCo '95)*, pages 279–285. Elsevier Science Publishers, September 1995.

[22] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages: Efficient, portable communication for workstation clusters and MPPs. *IEEE Concurrency*, 5(2):60–73, April 1997.

[23] L. Prylli and B. Tourancheau. BIP: a new protocol designed for high performance networking on Myrinet. In *1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW)*, volume 1388 of *Lect. Notes Comp. Science*, pages 472–485. IPPS/SPDP 1998, Springer-Verlag, April 1998.

[24] L. Prylli, B. Tourancheau, and R. Westrelin. An improved NIC program for high-perforamnce MPI. In *In Proc. of International Conference on Supercomputing (Workshop on Cluster-based Computing)*, pages 26–30, Rhodes, Greece, June 1999. ACM.

[25] R.D. Russell and P.J. Hatcher. Efficient kernel support for reliable communication. In *13th ACM Symposium on Applied Computing*, Atlanta, Georgia, February 1998.

[26] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proc. of 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, December 1995.

[27] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proc. 19th Int'l Symposium on Computer Architecture*, May 1992.

[28] T. von Eicken and W. Vogels. Evolution of the Virtual Interface Architecture. *IEEE Computer*, 31(11):61–68, November 1998.

[29] Boris Weismann, Benedict Gomes, Jurgen Quittek, and Michael Holtkamp. Efficient fine-grain thread migration with Active Threads. In *Proceedings of the First Merged Symposium of 12th Internation Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, pages 410–414, Orlando, Florida, March 1998. IEEE Computer Society.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399